

1. (20 pts.) Problem 1

- (a) This is false. A counterexample is $f(n) = 2n$ and $g(n) = n$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, since $2^{f(n)} = 2^{2n}$ and $2^{g(n)} = 2^n$, $\lim_{n \rightarrow \infty} \frac{2^{2n}}{2^n} = \infty \neq 0$. So, $2^{f(n)}$ grows faster than $2^{g(n)}$ asymptotically. Thus, the statement is false.
- (b) This is true. Proof: As $f(n)$ is $O(g(n))$, there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Then, $0 \leq f(n)^2 \leq c^2 g(n)^2$ for all $n \geq n_0$. Let $d = c^2$. We have $0 \leq f(n)^2 \leq dg(n)^2$ for all $n \geq n_0$. So, we've found constants d and n_0 that satisfy the definition of Big-O notation. Thus, $f(n)^2$ is $O(g(n)^2)$.
- (c) This is false. A counterexample is $f(n) = 2n$ and $g(n) = n$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Let $h(n)$ be $0.5n$. $h(n)$ is $O(g(n))$ in this case because we can find constants $c = 3$ and $n_0 = 1$ such that $0 \leq h(n) \leq cg(n)$ for all $n \geq n_0$. However, since $2^{f(n)} = 2^{2n}$ and $2^{h(n)} = 2^{0.5n}$, $\lim_{n \rightarrow \infty} \frac{2^{2n}}{2^{0.5n}} = \infty \neq 0$. So, $2^{f(n)}$ grows asymptotically faster than $2^{h(n)}$, which is an example of $2^{O(g(n))}$. Thus, the statement is false.
- (d) This is false. A counterexample is $f(n) = 2(1 + \frac{1}{n})$ and $g(n) = 1 + \frac{1}{n}$. $f(n)$ is $O(g(n))$ in this case because we can find constants $c = 2$ and $n_0 = 1$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. However, $\log_2 f(n) = \log_2(2(1 + \frac{1}{n})) = \log_2 2 + \log_2(1 + \frac{1}{n}) = 1 + \log_2(1 + \frac{1}{n})$ and $\log g(n) = \log_2(1 + \frac{1}{n})$. Note that $\lim_{n \rightarrow \infty} (1 + \frac{1}{n}) = 1$ implies that $\lim_{n \rightarrow \infty} \log(1 + \frac{1}{n}) = 0$. Any constant multiple $c \log g(n)$ also goes to 0 as n grows, while $\log f(n)$ always goes to 1 as n grows. Therefore, $\log f(n) \neq O(\log g(n))$.

2. (20 pts.) Problem 2

- (a) We can observe that term i with coefficient a_i has $\log_2 i$ multiplications. Overall, there are $\sum_{i=1}^n \log_2 i = \log_2(n!) = O(n \log n)$ multiplications (from worksheet 1 problem 2-i, $\log(n!) = \Theta(n \log n)$). There are n additions, thus $O(n)$.
- (b) If we consider the value of z after each iteration we obtain

$$\begin{aligned}
 i = n-1 & \rightarrow z = a_n x_0 + a_{n-1} \\
 i = n-2 & \rightarrow z = a_n x_0^2 + a_{n-1} x_0 + a_{n-2} \\
 i = n-3 & \rightarrow z = a_n x_0^3 + a_{n-1} x_0^2 + a_{n-2} x_0 + a_{n-3} \\
 & \dots \\
 i = 0 & \rightarrow z = a_n x_0^n + a_{n-1} x_0^{n-1} + \dots + a_1 x_0 + a_0,
 \end{aligned}$$

To describe in more detail, since the coefficients a_i are added to z in order from n to 0, the term with coefficient a_i multiplies with x_0 a total of i times. So, we have the desired polynomial $a_0 + a_1 x_0 + a_2 x_0^2 + \dots + a_n x_0^n$ at the end.

- (c) Every iteration of the for loop uses one multiplication and one addition, so the routine uses n additions and n multiplications.

3. (20 pts.) Problem 3

- (a) The while loop in algorithm 2 takes y iterations, and in each iteration we have to compute $z \cdot x$. Note that the multiplication of an n_1 bit number by an n_2 bit number results in a number with at most $n_1 + n_2$ bits. Therefore, at i^{th} iteration of the while loop, z has $O(in)$ number of bits (before multiplication), therefore the cost of multiplication at i^{th} iteration is $O(ni \log(ni))$, and since there are y iterations, the total running time would be:

$$\sum_{i=1}^y O(ni \log(ni)) = O(n) \sum_{i=1}^y i \log(ni) \quad (1)$$

$$= O(n) \left(\sum_{i=1}^y i \log n + \sum_{i=1}^y i \log i \right) \quad (2)$$

$$= O(n \log n) \sum_{i=1}^y i + O(n) \sum_{i=1}^y i \log i \quad (3)$$

$$= O(y^2 n \log n) + O(ny^2 \log y) \quad (4)$$

$$= O(n^2 y^2) \quad (5)$$

- (b) The function recurses until y becomes 0, dividing y by 2 each time. Therefore, there are $O(\log_2 y) = O(m)$ recursive calls. In each iteration, we either compute $z \cdot z$ or $z \cdot z \cdot x$. Both cases have a constant number of multiplications. Here, because i is $O(m)$ instead of $O(y)$, z has $O(n2^i)$ bits. Each multiplication then takes $O(n2^i \log(n2^i))$ time. Therefore, the total running time is:

$$\sum_{i=1}^m O(n2^i \log(n2^i)) = O(n) \sum_{i=1}^m 2^i \log(n2^i) \quad (6)$$

$$= O(n) \left(\sum_{i=1}^m 2^i \log n + \sum_{i=1}^m 2^i \log 2^i \right) \quad (7)$$

$$= O(n \log n) \sum_{i=1}^m 2^i + O(n) \sum_{i=1}^m i 2^i \quad (8)$$

$$\leq O(n \log n) \sum_{i=1}^m 2^i + O(mn) \sum_{i=1}^m 2^i \quad (9)$$

$$= O(2^m n \log n) + O(2^m mn) \quad (10)$$

$$= O(yn \log n) + O(yn^2) \quad (11)$$

$$= O(n^2 y) \quad (12)$$

Where step (10) uses the growth rate of the sum of a geometric series proven in homework 1 problem 3 and step (11) uses the stated assumption that $n \geq m$.

4. (20 pts.) Problem 4

- (a) For any 2×2 matrices X and Y :

$$XY = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix} = \begin{pmatrix} x_{11}y_{11} + x_{12}y_{21} & x_{11}y_{12} + x_{12}y_{22} \\ x_{21}y_{11} + x_{22}y_{21} & x_{21}y_{12} + x_{22}y_{22} \end{pmatrix}$$

This shows that every entry of XY is the addition of two products of the entries of the original matrices. Hence every entry can be computed in 2 multiplications and one addition. The whole matrix can be calculated in 8 multiplications and 4 additions.

- (b) Let $A' = XA$, where A is an arbitrary 2×2 matrix, and $X = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Since entries of A' are only the sum of at most two entries of A , then we can say that number of bits of A' entries are at most one bit more than number of bits in A . Therefore, each time we multiply a matrix by X , the number of bits of each entry only increases by at most one bit. Since all the entries of X can be stored in 1 bit, we can conclude that the entries of X^i have at most i bits. $i < n$ implies that the number of bits is $O(n)$.

Alternatively, we can use the fact that X^i stores the Fibonacci numbers: $X^i = \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix}$, which are known to grow more slowly than 2^i (see worksheet 2). The largest element, F_{i+1} has at most $O(\log_2(2^{i+1})) = O(i) = O(n)$ bits.

- (c) In each call of `matrix(X, n)`, we will go from n to $\frac{n}{2}$. So, it takes $\lfloor \log_2 n \rfloor$ recursive calls for the algorithm to end, and to return the output. Also, in each call, we have to do either $Z \cdot Z$ or $Z \cdot Z \cdot X$. Now, note that in i^{th} iteration or i^{th} recursive call we have $Z = X^{\frac{n}{2^i}}$, which means that entries of Z at i^{th} recursive call has at most $\frac{n}{2^i} = O(n)$ bits (According to part b). Now note that for computing either $Z \cdot Z$ or $Z \cdot Z \cdot X$, we have a constant number of multiplications and additions between numbers with $O(n)$ bits, which takes $O(M(n))$ time. Therefore, the total run time is:

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} O(M(n)) = O(M(n) \log n) \quad (13)$$

5. (20 pts.) Problem 5

- (a) The answer is at least 2^h and at most $2^{h+1} - 1$. This is because a complete binary tree of height $h - 1$ has $\sum_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is strictly larger than the number of vertices in a complete binary tree of height $h - 1$ and less than (or equal) the number of nodes in a complete binary tree of height h .
- (b) The array is not a Min heap. The node containing 26 is at position 9 of the array, so its parent is at position 4, which contains 35. This violates the Min Heap Property.
- (c) Consider the min heap with n vertices where the root and every other node contains the number 2. Suppose now that 1 is inserted to the first available position at the lowest level of the heap. That is, $A[i] = 2$ for $0 \leq i \leq n - 1$ and $A[n] = 1$. Since 1 is the minimum element of the heap, when `Heapify-UP` is called from position n , the node containing 1 must be swapped through each level of the heap until it is the new root node. Since the heap has height $\lfloor \log n \rfloor$, `Heapify-UP` has worst-case time $\Omega(\log n)$.

6. (0 pts.) Acknowledgments

- (a) I did not work in a group.
- (b) I did not consult with anyone other than my group members.
- (c) I did not consult any non-class materials.

Rubric:

Problem 1, total points 20

- (a) 5 points.
 - 2 point: correct conclusion (statement is false)
 - 1.5 points: a counterexample that makes sense
 - 1.5 points: an explanation of why the provided counterexample shows the statement is false
- (b) 5 points.
 - 2 point: correct conclusion (statement is true)
 - 3 points: a proof that reasons by making an association with the definition of big-O notation
- (c) 5 points.
 - 2 point: correct conclusion (statement is false)
 - 1.5 points: a counterexample that makes sense
 - 1.5 points: an explanation of why the provided counterexample shows the statement is false
- (d) 5 points.
 - 2 point: correct conclusion (statement is false)
 - 1.5 points: a counterexample that makes sense
 - 1.5 points: an explanation of why the provided counterexample shows the statement is false

Problem 2, total points 20

- (a) 5 points.
 - 1.5 points: correct answer for the number of sums
 - 1.5 points: correct answer for the number of multiplications
 - 2 points: the explanations make sense

Note: answers with exact numbers only or in big-O notations only are both acceptable, as long as the provided exact numbers or the big-O notations are correct.
- (b) 10 points.
 - 3 points: provide a proof
 - 7 points: describe the pattern of the loop in the proof
- (c) 5 points.
 - 1.5 points: correct answer for the number of sums
 - 1.5 points: correct answer for the number of multiplications
 - 2 points: the explanations make sense

Note: answers with exact numbers only or in big-O notations only are both acceptable, as long as the provided exact numbers or the big-O notations are correct.

Problem 3, 20 pts

- (a) 10 points. 6 points for showing the runtime of each iteration is $O(ni \log(ni))$. 4 points for manipulating the summation to show the overall runtime is $O(n^2 y^2)$.
- (b) 10 points.
 - 2 points: shows $O(m) = O(\log_2 y)$ recursive calls.
 - 4 points: shows the runtime of each recursive call is $O(n2^i \log(n2^i))$
 - 4 points: manipulating the summation to show the overall runtime is $O(n^2 y)$ (2 points for $O(n^3 y)$).

Problem 4, 20 pts

- (a) part a is worth 3 pts. Computing the multiplication of two matrices correctly is worth 2 pts, even if the final answer is not correct.
- (b) part b is worth 5 pts.
- (c) part c is worth 12 pts. 3 pts, for showing that there are $O(\log_2 n)$ recursive call, and 9 pts for showing the runtime for each recursive call which is $O(M(\frac{n}{2^i}))$. However, $O(M(n))$ is also accepted for the runtime of each recursive call.

Problem 5, 20 pts.

- (a) 5 points for this part, if at least 2^h is pointed, 1.5 pts, if at most $2^{h+1} - 1$ is pointed, 1.5 pts, reasonable explanation 2 pts
- (b) answer as No, 2.5 pts, denote that element 26 is in the wrong place, and it should be swapped with 35, 2.5 pts
- (c) any clear and reasonable explanation should be 10 pts